CSC 311 – Winter 2022-2023
Design and Analysis of Algorithms

7. Dynamic Programming

Prof. Mohamed Menai
Department of Computer Science
King Saud University

Outline

- Dynamic programming
- Longest Common Subsequence problem
- 0-1 Knapsack problem

Dynamic Programming

- Another strategy for designing algorithms is *dynamic programming (DP)*
  - A technique, not an algorithm (like divide-and-conquer)
  - Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
  - The word "programming" is historical and predates computer programming
- DP can be applied when the solution of a problem includes solutions to subproblems

Dynamic Programming

- We need to find a recursive formula for the solution
- We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory (Table)
- In the end we will get the solution of the whole problem
- More efficient than *brute-force methods*, which solve the same subproblems over and over again

Properties of a problem that can be solved with DP

- Simple Subproblems
  - We should be able to break the original problem to smaller subproblems that have the same structure
- Optimal Substructure of the problems
  - The solution to the problem must be a composition of subproblem solutions
- Subproblem Overlap
  - Optimal subproblems to unrelated problems can contain subproblems in common

Example: Fibonacci numbers

- Recall definition of Fibonacci numbers:

  F(n) = F(n-1) + F(n-2)
  F(0) = 0
  F(1) = 1

- Computing the nth Fibonacci number recursively (top-down):

F(n)

F(n-1) + F(n-2)

F(n-2) + F(n-3) F(n-3) + F(n-4)

...

6

Example: Fibonacci numbers

Computing the nth Fibonacci number using bottom-up iteration and recording results:

F(0) = 0
F(1) = 1
F(2) = 1+0 = 1
…
F(n-2) =
F(n-1) =
F(n) = F(n-1) + F(n-2)

0	**1**	**1**	**. . .**	***F*(*n*-2)**	***F*(*n*-1)**	***F*(*n*)**

Efficiency?
- time
- space

DP Example: Longest Common Subsequence (LCS)

- *Longest common subsequence (LCS)* problem:
  - Given two sequences x[1..m] and y[1..n], find the longest subsequence which occurs in both
  - Ex: X = {A B C B D A B }, Y = {B D C A B A}
  - {B C} and {A A} are both subsequences of both sequences
    - *What is the LCS?*
  - Brute-force algorithm: For every subsequence of X, check if it is a subsequence of Y
    - 2^m subsequences of X to check against n elements of Y: $O(n.2^m)$

Longest Common Subsequence (LCS)

- DP algorithm: solve subproblems until we get the final solution
- Subproblem: first find the LCS of *prefixes* of X and Y.
- This problem has *optimal substructure*: LCS of two prefixes is always a part of LCS of bigger strings

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: X= {A B C B D A B }, Y= {B D C A B A}

Longest Common Subsequence:

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

LCS Algorithm

- First we will find the length of LCS. Later we will modify the algorithm to find LCS itself.
- Define $x[i]$, $y[j]$ to be the prefixes of X and Y of length i and j, respectively
- Define $c[i,j]$ to be the length of LCS of $x[i]$ and $y[j]$
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

LCS recursive solution

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of *x* and *y*)
- Since *x*[0] and *y*[0] are empty strings, their LCS is always empty (i.e. *c*[0,0] = 0)
- LCS of empty string and any other string is empty, so for every *i* and *j*: *c*[0, *j*] = *c*[*i*,0] = 0

LCS recursive solution

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

- When we calculate $c[i,j]$, we consider two cases:
- **First case:** $x[i]=y[j]$
  - One more symbol in strings X and Y matches, so the length of LCS of $x[i]$ and $y[j]$ equals to the length of LCS of smaller strings $x[i-1]$ and $y[j-1]$, plus 1

LCS recursive solution

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- **Second case:** $x[i]$!= $y[j]$
  - Symbols don't match and the length of LCS($x[i]$, $y[j]$) is the maximum of LCS($x[i]$, $y[j$-1]) and LCS($x[i$-1], $y[j]$)
- Why not just take the length of LCS(X_{i-1}, Y_{j-1}) ?

14

LCS recursive solution

X=abc and Y=db

c[3,2]=max(c[3,1],c[2,2])=max(0,1)=1

whereas c[3,2] != c[2,1]=0

LCS Length Algorithm

```
LCS-Length(X, Y)
1. m = length(X)  // get the # of symbols in X
2. n  = length(Y) // get the # of symbols in Y
3. for i = 0 to m       c[i,0] = 0       // special case: Y_0
4. for j = 0 to n       c[0,j] = 0       // special case: X_0
5. for i = 1 to m                        // for all X_i
6.        for j = 1 to n                           // for all Y_j
7.                if ( X_i == Y_j )
8.                        c[i,j] = c[i-1,j-1] + 1
9.                else c[i,j] = max(c[i-1,j], c[i,j-1])
10. return c
```

16

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB
- What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS Example (0)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,6]

LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 0 to m   c[i,0] = 0
for j = 0 to n   c[0,j] = 0

LCS Example (2)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		B	D	C	A	B
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	A		**0**	**0**				
2	B		**0**					
3	C		**0**					
4	B		**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

20

LCS Example (3)

ABCB
BDCAB

i		j: 0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

21

LCS Example (4)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

22

LCS Example (5)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (6)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1				
3	C		0					
4	B		0					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

25

LCS Example (8)

ABCB
BDCAB

i	Xi \ j	0 (Yj)	1 (B)	2 (D)	3 (C)	4 (A)	5 (B)
0		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (10)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**			
4	**B**	**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

27

LCS Example (11)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (12)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

29

LCS Example (13)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**				

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

30

LCS Example (14)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (15)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**	**1**	**2**	**2**	**3**

if ($X_i == Y_j$)
 c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

32

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array $c[m,n]$
- So what is the running time?

O($m.n$) since each $c[i,j]$ is calculated in constant time, and there are $m.n$ elements in the array

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each $c[i,j]$ depends on $c[i\text{-}1,j]$ and $c[i,j\text{-}1]$

or $c[i\text{-}1,j\text{-}1]$

For each $c[i,j]$ we can say how it was acquired:

2	2
2	3

For example, here

$c[i,j] = c[i\text{-}1,j\text{-}1] +1 = 2+1=3$

How to find actual LCS

- Remember that

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- So we can start from $c[m,n]$ and go backwards
- Whenever $c[i,j] = c[i\text{-}1, j\text{-}1]+1$, remember $x[i]$ (because $x[i]$ is a part of LCS)
- When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1** ←	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2** ←	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

36

Finding LCS (2)

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1** ←	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2** ←	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

LCS (reversed order): **B C B**

LCS (straight order): **B C B**

37

Knapsack problem

- Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W.
- So we must consider weights of items as well as their value.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

38

Knapsack problem

- There are two versions of the problem:
  - (1) "0-1 knapsack problem" and
  - (2) "Fractional knapsack problem"

(1) Items are indivisible; you either take an item or not. Solved with DP

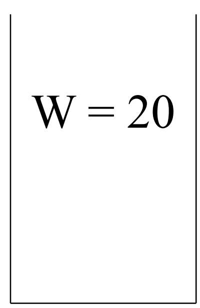
(2) Items are divisible: you can take any fraction of an item. Solved with a greedy algorithm

0-1 Knapsack problem

- Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
- Each item *i* has some weight w_i and benefit value b_i (all w_i , b_i and *W* are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem: a picture

This is a knapsack
Max weight: W = 20



Items	Weight w_i	Benefit value b_i
	2	3
	3	4
	4	5
	5	8
	9	10

41

0-1 Knapsack problem

- Problem, in other words, is to solve:

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a "0-1" problem, because each item must be entirely accepted or rejected.
- The other version of this problem is the "*Fractional Knapsack Problem*", where we can take fractions of items.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$

Defining a Subproblem

If items are labeled 1..*n*, then a subproblem would be to find an optimal solution for S_k = {*items labeled* 1, 2, .. *k*}

- This is a valid subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we <u>cannot</u> do that. Explanation follows….

Defining a Subproblem

$w_1 = 2$ $b_1 = 3$	$w_3 = 4$ $b_3 = 5$	$w_4 = 5$ $b_4 = 8$	$w_2 = 3$ $b_2 = 4$	

?

Max weight: W = 20

For S_4:
Total weight: 14;
total benefit: 20

$w_1 = 2$ $b_1 = 3$	$w_3 = 4$ $b_3 = 5$	$w_4 = 5$ $b_4 = 8$	$w_5 = 9$ $b_5 = 10$

For S_5:
Total weight: 20
total benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

S_4

S_5

Solution for S_4 is not part of the solution for S_5!

Defining a Subproblem

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: w, which will represent the exact weight for each subset of items
- The subproblem then will be to compute $B[k,w]$

Recursive Formula for subproblems

- Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- It means that the best subset of S_k that has total weight w is one of the two:

  1) the best subset of S_{k-1} that has total weight w, **or**

  2) the best subset of S_{k-1} that has total weight w-w_k plus the item k

Recursive Formula

$$B[k,w]=\begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- **First case**: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be > w, which is unacceptable
- **Second case**: $w_k <= w$. Then the item k can be in the solution, and we choose the case with greater value

48

0-1 Knapsack Algorithm

```
for w = 0 to W
   B[0,w] = 0
for i = 0 to n
   B[i,0] = 0
   for w = 1 to W
        if w_i <= w // item i can be part of the solution
                if b_i + B[i-1,w-w_i] > B[i-1,w]
                        B[i,w] = b_i + B[i-1,w- w_i]
                else
                        B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
```

Running time

```
for w = 0 to W
   B[0,w] = 0
for i = 0 to n
   B[i,0] = 0
   for w = 0 to W
       < the rest of the code >
```

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

$O(n.W)$

Remember that the brute-force algorithm takes $O(2^n)$

50

Example

Let’s run the algorithm on the following data:

n = 4 (# of elements)

W = 5 (max weight)

Elements (weight, benefit):

(2,3), (3,4), (4,5), (5,6)

Example (2)

W \ i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				

for w = 0 to W

B[w,0] = 0

Example (3)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0				
2	0				
3	0				
4	0				
5	0				

```
for i = 0 to n
      B[0,i] = 0
```

Example (4)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0 →	0			
2	0				
3	0				
4	0				
5	0				

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=1$
$b_i=3$
$w_i=2$
$w=1$
$w-w_i=-1$

```
if wi <= w // item i can be part of the solution
    if bi + B[i-1,w-wi] > B[i-1,w]
        B[i,w] = bi + B[i-1,w- wi]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // wi > w
```

54

Example (5)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	**3**			
3	0				
4	0				
5	0				

$i=1$
$b_i=3$
$w_i=2$
$w=2$
$w-w_i=0$

if $w_i <= w$ // item i can be part of the solution
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w-w_i]$
  else
    $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example (6)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	**3**			
4	0				
5	0				

$i=1$
$b_i=3$
$w_i=2$
$w=3$
$w-w_i=1$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

56

Example (7)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	**3**			
5	0				

$i=1$
$b_i=3$
$w_i=2$
$w=4$
$w-w_i=2$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (8)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	3			
3	0	3			
4	0	3			
5	0	**3**			

$i=1$
$b_i=3$
$w_i=2$
$w=5$
$w-w_i=2$

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

58

Example (9)

w \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0 →	**0**		
2	0	3			
3	0	3			
4	0	3			
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

59

Example (10)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	→ **3**		
3	0	3			
4	0	3			
5	0	3			

i=2
b_i=4
w_i=3
w=2
$w-w_i$=-1

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // w_i > w

60

Example (11)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	**4**		
4	0	3			
5	0	3			

$i=2$
$b_i=4$
$w_i=3$
$w=3$
$w-w_i=0$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

61

Example (12)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	**4**		
5	0	3			

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

62

Example (13)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	4		
4	0	3	4		
5	0	3	**7**		

$i=2$
$b_i=4$
$w_i=3$
$w=5$
$w-w_i=2$

if $w_i <= w$ // item i can be part of the solution
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w- w_i]$
  else
    $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

63

Example (14)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0 →	**0**	
2	0	3	3 →	**3**	
3	0	3	4 →	**4**	
4	0	3	4		
5	0	3	7		

$i=3$
$b_i=5$
$w_i=4$
$w=1..3$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // w_i > w
```

Example (15)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	**5**	
5	0	3	7		

$i=3$
$b_i=5$
$w_i=4$
$w=4$
$w- w_i=0$

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_i + B[i-1,w- w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

65

Example (15)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	→ **7**	

$i=3$
$b_i=5$
$w_i=4$
$w=5$
$w- w_i=1$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

Example (16)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0 →	**0**
2	0	3	3	3 →	**3**
3	0	3	4	4 →	**4**
4	0	3	4	5 →	**5**
5	0	3	7	7	

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

$i=4$
$b_i=6$
$w_i=5$
$w=1..4$

if $w_i <= w$ // item i can be part of the solution
  if $b_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = b_i + B[i-1,w- w_i]$
  else
    $B[i,w] = B[i-1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$

67

Example (17)

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

W \ i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	4	5	5
5	0	3	7	7 $\longrightarrow$	**7**

$i=4$
$b_i=6$
$w_i=5$
$w=5$

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w

68

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary
- Running time (Dynamic Programming algorithm vs. brute-force algorithm):
  - LCS: $O(m.n)$ vs. $O(n.2^m)$
  - 0-1 Knapsack problem: $O(W.n)$ vs. $O(2^n)$

Reading

Chapter 8

Anany Levitin, Introduction to the design and analysis of algorithms, 3rd Edition, Pearson, 2011.